MPI-IO/L: Efficient Remote I/O for MPI-IO via Logistical Networking

Jonghyun Lee* Robert Ross* Scott Atchley† Micah Beck‡ Rajeev Thakur*

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

†Myricom, Inc., Oak Ridge, TN 37830, U.S.A.

‡Department of Computer Science, University of Tennessee, Knoxville, TN 37996, U.S.A.

{jlee, rross, thakur}@mcs.anl.gov, atchley@myri.com, mbeck@cs.utk.edu

Abstract

Scientific applications often need to access remotely located files. However, many remote I/O systems lack standard APIs that allow efficient and direct access from application codes. This work presents MPI-IO/L, a remote I/O facility for MPI-IO by using Logistical Networking. This combination not only provides high-performance and direct remote I/O using the standard parallel I/O interface but also offers convenient management and sharing of remote files. We show the performance tradeoffs between various remote I/O approaches implemented in the system, which can help scientists identify preferable I/O options for their own applications. We also discuss how Logistical Networking could be improved to work better with parallel I/O systems such as ROMIO.

1 Introduction

Scientists are running an increasing number of applications in distributed environments, with the frequent need to store and retrieve data at remote locations. For example, a scientist running a simulation at a remote supercomputing center might wish to store the data locally to visualize it on her own machine. Also, a group of collaborating scientists geographically separated from one another might wish to set up a central data repository for efficient data sharing. Further, the emergence of Grid computing [8] enables the execution of high-performance distributed applications, which might transfer data among their components through a remote I/O mechanism.

Traditionally, scientists have performed remote I/O by temporarily *staging* remote files on local disks. Although staging seeks to boost the I/O performance by collocating data with the application, it imposes several problems. First, since it does not allow applications to directly access remote files, it causes extra disk I/O. Second, since each file is staged in its entirety, excessive transfer occurs when only a portion of the file is needed. Third, it requires enough space on the local disk to hold the staged files. Fourth, consistency problems can arise if the remote file has been modified at the source after it has been staged locally. Fifth, staging is often done manually and thus is cumbersome.

A better remote I/O system should address the following I/O and file management needs typically required by many scientists.

Functionality. Direct access to any portion of remote files through a convenient interface should be possible. Since many scientific codes are parallel, supporting parallel I/O is highly beneficial.

Performance. Remote I/O is often slow because of the low Internet bandwidth and the amount of data to be accessed. It is important to reduce apparent remote I/O cost by using high-performance data transfer mechanisms and optimizations [2, 10, 13].

Management. Scientific data files are often replicated or striped across multiple remote storages for fault tolerance, a faster access or because of storage constraints. I/O to such "distributed" files requires detailed information about each replica or stripe (e.g., its physical location and mapping to the logical file). Manually maintaining such information can be cumbersome, especially with a number of logical files. It can be managed better by an I/O system.

Sharing. Scientific data is often shared among a group of people. Giving out the information about the files to others and accessing the files using such information should be easy. Also, file access should be limited to authorized users only. It is desirable for an I/O system to support efficient and secure sharing.

This work addresses the above issues by integrating a parallel I/O library with an efficient and flexible remote I/O functionality. We chose the ROMIO [15] implementation of MPI-IO [12] for the testbed I/O library and the Logistical Networking software [5] for the remote I/O and file management capability. MPI-IO is the de facto parallel I/O interface standard, used both directly by applications and by high-level libraries such as Parallel NetCDF [11] and HDF5 [1]. Supporting remote I/O through MPI-IO thus enables many applications to perform remote I/O transparently without code

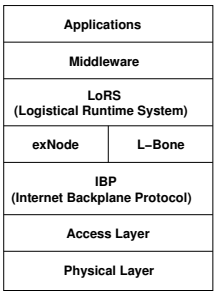


Figure 1. Network storage stack.

changes. Logistical Networking provides powerful remote I/O mechanisms, including high-performance data transfer by multiple concurrent streams and intelligent download schemes [13]. It also flexibly describes the relationship between a logical file and its associated physical files with XML, thereby easing the sharing (Section 2).

The contributions of this work are as follows. First, we identify the design issues for a Logistical Networking-based remote I/O facility for a parallel I/O system and provide an implementation with ROMIO, called MPI-IO/L. Second, we optimize basic I/O operations for MPI-IO/L and discuss the tradeoffs of the proposed approaches, helping users select the preferable remote I/O options for their applications. Third, we identify ways to make Logistical Networking work better with parallel I/O systems such as ROMIO.

2 Logistical Networking

The goal of Logistical Networking is to provide scalable and sharable storage resources and services that increase the efficiency, performance, and functionality of distributed applications. The center of this technology is the Network Storage Stack (Figure 1), through which writable storage can be added to the Internet as a network resource. It was modeled after the Internet Protocol (IP) stack. At the bottom is the physical layer, which includes storage media such as disk, memory, or tape. Next is the access layer, the storage service provided by the operating system such as disk driver, analogous to the link layer in the IP stack. The middle layers are the core of Logistical Networking and consist of IBP, L-Bone, exNode, and LoRS. Each is described below in detail. On top of them sits the middleware layer, which provides services using underlying components (e.g., ROMIO, Parallel netCDF, and HDF5). At the top is the application layer.

IBP. The Internet Backplane Protocol (IBP) allows anyone to share disk or memory space on her machine over the network. Unlike FTP servers, these IBP *depots* do not maintain user accounts, file system hierarchies, and the like. A client can request an allocation (a specific amount of space) for a specific amount of time to any depots and can access and share the allocation until it expires. Each depot is preset with the total amount of storage space that it is willing to share and the maximum time allowed for any single allocation.

IBP was designed to provide a simple, yet highly scalable storage service and thus implements only the following operations. The operation `allocate()` either grants or denies client allocation requests based on the depot's currently available space and the maximum duration policy. For a granted request, the depot sends a set of capabilities (keys) to the client, which are needed for write, read, and manage access (described below) to the allocation. If a client wishes to share an allocation with others, she can give them a subset of capabilities (e.g., read only) associated with it and control their access to the allocation. The operations `store()` and `load()` write to and read from an allocation. While read can begin from any offset within the allocation, writes are currently append only. The operations `copy()` and `mcopy()` perform third-party transfers between IBP depots, and the operation `manage()` changes the properties of an allocation, such as size and duration.

L-Bone. To access depots, clients need the hostnames and their port numbers. The Logistical Backbone (L-Bone) helps users find available public depots and their associated information. L-Bone is an LDAP-based server that catalogs available depots and polls them periodically to determine available storage. In addition to the hostname, port, allocation duration policy, and recent space availability, L-Bone can also store information regarding each depot, such as its geographical location. Clients can send a request to an L-Bone server for a list of depots that meet specific criteria.

ExNode. A client may need to allocate more space than any single depot may have (or is willing to give a single allocation request) or may need to create replicas on different depots. Either case requires having multiple allocations and capabilities for a logical file. The exNode is a data structure that facilitates handling of multiple allocations by aggregating and annotating them.

ExNodes resemble Unix inodes in that just as an inode aggregates individual disk blocks to create large files, an exNode aggregates IBP allocations and provides a mapping from the logical view of a file to the allocations that store the actual data. Unlike the inode, however, the exNode allows for varying size allocations and for replication of data in different allocations. Also, an inode has fixed metadata, while the exNode allows for arbitrary metadata for the global exNode as well as for mappings to individual IBP allocations. The exNode also allows for a function that describes whether the stored data was encoded before being stored (e.g., encrypted, compressed). The exNode can be serialized to XML for sharing between processes or clients.

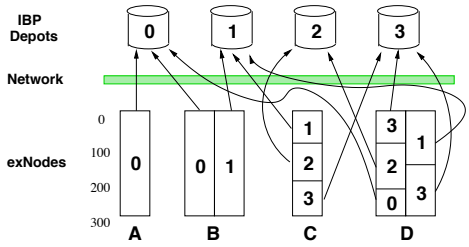


Figure 2. Sample exNodes.

```
<exnode:mapping>
  <exnode:metadata name="alloc_length" type="integer">100</exnode:metadata>
  <exnode:metadata name="alloc_offset" type="integer">0</exnode:metadata>
  <exnode:metadata name="exnode_offset" type="integer">100</exnode:metadata>
  <exnode:metadata name="logical_length" type="integer">100</exnode:metadata>
  <exnode:read>ibp://depot2:[port]/[key string]/READ</exnode:read>
  <exnode:write>ibp://depot2:[port]/[key string]/WRITE</exnode:write>
  <exnode:manage>ibp://depot2:[port]/[key string]/MANAGE</exnode:manage>
</exnode:mapping>
```

Figure 3. A serialized mapping in XML.

Figure 2 shows the exNodes for four logical files, each 300 bytes long. The numbers shown in each exNode indicates which depot stores an allocation for a certain portion of each file. Here, A is stored in an allocation on the depot 0, while B is replicated on the depots 0 and 1. C is striped over three depots. D is both replicated and striped, where the first replica is striped over three depots and the second one is striped over two depots. Figure 3 shows the serialized exNode in XML for a mapping to an allocation of C stored in the depot 2.

LoRS. The Logistical Runtime System (LoRS) provides APIs and command line tools that automate the finding of depots via the L-Bone, creating and using allocations and capabilities and creating exNodes. The command line tools allow the user to upload, download, augment (add replicas), trim (remove replicas), refresh (renew IBP leases) and list (check the exNode's status) *network files*.[1] The APIs are much more flexible than the command line tools and can be used to build a more complex system such as the facility described here.

3 Design

MPI-IO/L exploits the intermediate ADIO (Abstract Device I/O) [16] layer in ROMIO. ADIO defines a set of basic I/O interfaces, which can be used to implement more complex, higher-level I/O interfaces such as MPI-IO. For each supported file system, ADIO requires a separate implementation (a "module") of its I/O interfaces, and a new remote I/O-specific ADIO module was added to ROMIO for MPI-IO/L. Figure 4 depicts this architecture. The bold arrows show the data flow for a write operation to a remote storage through MPI-IO/L.

3.1 Network File Access from MPI-IO

MPI-IO provides the standard interface for parallel I/O from multiple processes onto a common logical file. Thus, MPI-IO implementations such as ROMIO assume processes in a run can access a shared file system. Each process can access the file either cooperatively with other processes (collective I/O[2]) or independently, and multiple processes can concurrently operate on any regions of the file.

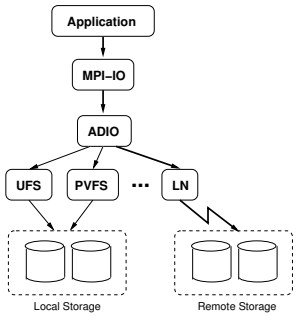


Figure 4. ROMIO's layered architecture.

When multiple processes are working on a common file, the file attributes (e.g., the mapping between logical and physical file contents) need to be shared among the processes, and any process that accesses the file can modify this *metadata*. Local file systems store the metadata in an internal data structure such as the inode at the same storage where the actual file data resides. Logistical Networking, however, maintains the metadata, contained in the exNode, locally where the application runs, while the file data can be stored remotely. Also, according to the MPI-IO consistency semantics (Section 3.3), each file data and metadata update need not be immediately visible to other processes unless explicit synchronization methods are used. Accordingly, in MPI-IO/L, each process keeps a separate exNode in its memory and only locally updates it for each write. At the user's request (by calling sync or close), the exNodes are synchronized among processes, and thus the changes made by other processes become visible.

3.2 MPI-IO/L ADIO Module

Our original plan for MPI-IO/L was to use libxio,[3] which provides standard Unix I/O interfaces (e.g., open, close, read, and write) for network files. MPI-IO/L could be built by replacing Unix I/O calls in the UFS (Unix File System) ADIO module, with their libxio equivalents. But, libxio is not yet complete and was not specifically designed or optimized for parallel I/O. Thus, we wrote our own ADIO functions, partly based on libxio version 0.2, as described below.

Open and Close. `LN_Open` and `LN_Close` collectively opens and closes a network file from multiple processes. Typically, a file open is called with the file name, and then its associated metadata is either cre-

[1] We use the term *network file* to refer a file created by using Logistical Networking.

[2] Collective I/O optimizes parallel I/O operations by using the global knowledge of data distribution in memory and file [15].

[3] http://loci.cs.utk.edu/lors/distributions/libxio-0.2.tar.gz

ated or located. Since the metadata of a network file is not likely to be collocated with the file data, however, `LN_Open` is called with the name of an XML exNode file along with other attributes such as file access mode.[4] MPI-IO/L assumes that the exNode file will be accessed from a local file system and has only the root process (with the rank 0) read and write the file.

When called, `LN_Open` creates two data structures in each process's memory. First, it creates an in-memory exNode that contains allocation- and mapping-related information for the network file. If the given XML file exists, the root first reads it into memory and broadcasts its content to the other processes. Then, each process creates an empty exNode in its memory and *deserializes* the XML file content into it. If the file does not exist, each process creates only an empty exNode.

Next, `LN_Open` creates a *depotpool* on each process that contains a list of depots that will be used for data storage and retrieval. If a network file is open for read only, the depotpool is created by extracting from the XML file all the depots that provided allocations for previous writes. If the file is open for write only, an L-Bone server is contacted to find a set of depots that satisfy the user's storage requirements, such as the size and duration of available storage space, the number of depots to be found, and location preference. Such information can be passed to `LN_Open` as hints by using an `MPI_Info` object. The user can also provide a list of known depots that she wants to use. LoRS provides separate APIs for the depotpool creation for read-only and write-only cases.

When a file is open for read and write, both depot extraction and search should be performed. However, LoRS currently does not provide a single API for both operations. Moreover, it does not allow the two APIs used for read-only and write-only cases to be combined. MPI-IO/L deals with this problem by manually extracting depots from the XML file and adding them to the L-Bone search results. This is not a desired solution, however, because LoRS maintains only one depotpool, and thus the depots extracted for future reads are included in the same depotpool that contains the ones selected for writes. Hence, the read depots might be used for future write operations, and if they fail to meet the user's storage requirements, an error will occur. A better approach is to keep two separate depotpools for reads and writes and constantly update the read depotpool whenever a new depot is used for a write. A future version of LoRS should consider this option.

When each process contacts an L-Bone server for writable files, it is likely to be given the identical list of depots because each sends the same storage requirements to the L-Bone server. Since LoRS contacts the depots for allocations in the order they appear in the depotpool, if multiple processes request allocations at the same time, all the write requests will be sent to the first depot in the list. With multiple depots available in the depotpool, performance is often better if we distribute the write requests to multiple depots and thus trigger concurrent writes to them. MPI-IO/L reorders the depots in the depotpool by default to distribute concurrent write operations to different depots.

Other hints can be passed at file open. They include the hostname and port number of a preferred L-Bone server, the size of the unit data transfer (called *block*), the number of replicas to create at each write, the number of threads to be used for concurrent transfers, the size of the memory buffer for buffered I/O (described later), and the options for MPI-IO/L specific optimizations. After creating the two data structures on each process, `LN_Open` initializes internal variables according to the passed hint values, allocates buffer memory if buffered I/O is enabled, and then returns.

`LN_Close` synchronizes exNodes from all the processes, has the root *serialize* the synchronized exnode to the XML file, frees internal data structures, and returns. More on this is presented in Section 3.3.

Contiguous I/O. MPI-IO/L provides two modes of contiguous I/O. *Direct I/O* issues remote I/O requests immediately. *Buffered I/O* temporarily buffers the write data using preallocated memory. The buffer holds one contiguous subextent of the file, whose coverage can change dynamically. If the write data cannot be buffered in the current buffer in its entirety or will not form a contiguous extent when combined with previously buffered data, the buffer is flushed by a direct write before buffering the new data. If the write data is larger than the size of the allocated buffer, only a buffer size of data at the end is buffered; the rest is written directly. Buffered read is carried out from the buffer if it contains at least a portion of the the requested data and the nonbuffered portion is read directly.

Buffered I/O is not asynchronous because LoRS does not have asynchronous I/O interface yet. It is rather a write optimization to coalesce several small writes into few large writes to reduce both per-operation overhead and the number of allocations and mappings created.[5] It can reduce the number of the actual I/O operations to be performed if a certain buffered region is overwritten or read multiple times. If asynchronous I/O becomes available in LoRS, buffered I/O can further improve I/O performance by write-behind and prefetching.

`LN_WriteContig` first checks the current I/O mode. Direct write is chosen when the buffer size is set to 0. Otherwise, buffered write is performed,

[4] Following ROMIO's file naming convention, the prefix "`ln:`" should be used in front of the XML file name.

[5] For each write request, LoRS currently creates a separate allocation and a mapping to it.

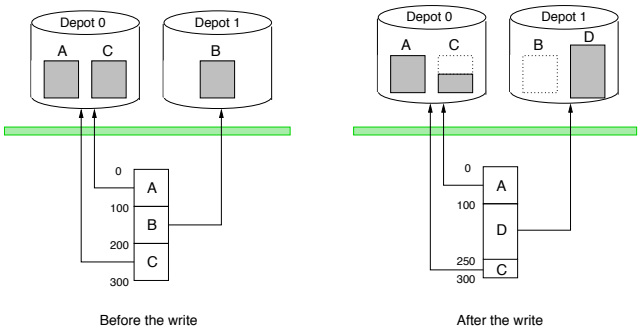


Figure 5. Status of an exNode before and after "Write 150 bytes at offset 100."

which in turn can issue direct writes. Direct write is performed by the LoRS function, `lorsSetStore()`, which creates a new allocation, stores the data, creates mappings to it, and places the mapping into the set.[6] If the data is larger than the block size, it is further divided to issue smaller writes.

The write to an unwritten portion of the file is performed by calling `lorsSetStore()` and adding the newly created mapping(s) to the exNode. However, an overwrite over existing mappings goes through the following steps. First, the mappings that overlap with the write request are removed from the exNode and put into a set. Second, each mapping in the set is either removed if the logical extent that it points to will be completely overwritten or trimmed if it will be partially overwritten. Third, `lorsSetStore()` is called to write the requested data and create new mappings to new allocations in another set. Fourth, the two sets are merged, and the mappings in the merged set are added to the exNode. Figure 5 shows how an exNode is changed after an overwrite. The original exNode contains three mappings to three allocations (A, B, and C) across two depots for a 300-byte-long logical file. When an operation that writes 150 bytes at offset 100 is issued, the mappings to B and C overlap with the write request, and they are either removed or trimmed accordingly. It also writes the data to a new allocation D and updates the exNode. In the figure, only the shadowed portion of each allocation is mapped to a portion of the logical file. For example, after the write, only the second half of C is mapped to the file. B is not mapped to the file any more but remains until it expires or is revoked explicitly. If the depots supported write-at-offset, the overwrite could be performed without additional allocation and the change of mappings in the exNode.

`LN_ReadContig` also checks the I/O mode first. If direct read is set, mappings that overlap with the requested extent are identified and put into a set but are not removed from the exNode this time. If the mappings in the set form a contiguous extent, `lorsSetLoad()` is called once for the set, which reads the allocations pointed to by the mappings in the set. If holes[7] are formed by unwritten portions in the extent, however, the current implementation of `lorsSetLoad()` will return an error when attempting to read that extent. To get around this problem, it first identifies each contiguous region in the whole extent and which mappings will form the region. Then, it issues a separate `lorsSetLoad()` for each contiguous region and combines the results in the user buffer.

Noncontiguous I/O. Noncontiguous I/O can be performed in a few ways. A naïve approach is to issue a separate I/O request for each contiguous region contained in the extent, but this will incur overhead for each I/O operation issued. ROMIO optimizes noncontiguous I/O through data sieving [15]. For a noncontiguous read, data sieving reads the whole extent of the I/O request and picks out only the regions that are supposed to be read. For a noncontiguous write, data sieving reads the whole extent of the request into a buffer, modifies the regions to be written in the buffer, and writes the whole buffer back. Since the whole procedure must be performed atomically, however, this approach works only with file systems that provide file locking, making it unsuitable for MPI-IO/L.

MPI-IO/L provides two different options for noncontiguous writes. First, it simply reuses ROMIO's naïve noncontiguous write routine, which in turn makes separate direct write calls for each contiguous region in the extent. Second, MPI-IO/L optimizes noncontiguous writes by exploiting the fact that multiple mappings can be flexibly generated to a single IBP allocation. For this, the data that will be written to the noncontiguous extent is first packed into a contiguous memory buffer. Then, the packed data buffer is written into a single allocation or a small number of allocations if the block size is smaller than the size of the packed data. Next, a separate mapping is created between each contiguous region in the original extent and its corresponding portion in the newly created allocation in the exNode. If a contiguous region is stored across the boundaries of two allocations, two separate mappings must be created for each allocation. This method resembles a log-structured file system in that it stores noncontiguous regions contiguously. However, LoRS currently does not provide a single API that creates multiple arbitrary mappings to a single allocation (`lorsSetStore()` creates only one mapping per allocation). MPI-IO/L gets around this problem by calling `lorsSetStore()` once to store the packed data, memory copying the newly created mapping(s) multiple times, modifying the offset and length fields

[6] In LoRS, a set is defined as a collection of mappings, on which a common operation can be executed in batch.

[7] We distinguish two types of holes. For the holes resulted from unwritten portions of the file, we treat them as the regions filled with unknown values. But, if holes are formed by already expired allocations, reading such region is considered to be an error.

for both logical and physical allocation in each copied mapping accordingly, and adding the modified mappings, not the ones created by `lorsSetStore()`, to the exNode. All these procedures are performed in memory. A future version of LoRS should include a single API that does the same job.

The second approach is expected to perform better than the naïve approach because it can significantly reduce the number of remote write calls and thus the overhead associated with each invocation. Moreover, the manipulation of mappings is done in memory and is not expected to be time-consuming. However, it will create the same number of mappings as (or even more mappings in some cases than) the naïve approach. Also, even though a set of contiguous regions is packed and stored in a single allocation, future reads whose extents overlap with the noncontiguous write extent cannot recognize it, and thus be optimized, because the exNode currently does not provide an efficient way to describe such information. User can choose which write option to use by providing a hint at file open. The default is the optimized approach.

MPI-IO/L also provides two options for noncontiguous read. As in noncontiguous write, each contiguous region in the extent can be read by a separate read call. Alternatively, data sieving can be used, because data sieving read does not require locking. Both approaches have trade-offs. While the naïve approach involves many more remote read calls and thus is likely to incur higher overhead, data sieving causes extra data access, which could be expensive for remote file systems. Careful performance study is needed to decide which to choose for a given noncontiguous read request. The user can choose the method of noncontiguous reads by providing a hint at file open. The default is data sieving.

Since noncontiguous I/O implementations eventually call file system-specific contiguous I/O calls, buffered I/O can still be used, although each buffer is supposed to hold a contiguous extent and thus is not of much help for noncontiguous I/O. The only exception is the optimized noncontiguous write, which does not call contiguous write routine. In this case, if the current buffer extent overlaps with the extent of the noncontiguous write request, we simply flush the buffer before taking actions for the noncontiguous write.

Other Functions. A sync operation causes all previous writes to be transferred to the storage device so that the changes (both to file data and to metadata) made by one process will be visible from other processes. However, IBP currently does not provide a mechanism for a file data sync. `lorsSetStore()` returns only after the data has been written to a local file system on the depot. Therefore, in case of a depot crash, the data stored in the file cache might be lost. For this reason, `LN_Sync` synchronizes exNodes only among processes, which is described in detail later.

`LN_Delete` is implemented by just deleting the locally stored XML file. The allocations used in the XML file will be revoked when their expiration occurs. A more proactive approach would be freeing the allocations at the time of delete, but this could cause problems if some of the allocations contained in the XML file were used in other exNodes.

`LN_Resize` is implemented in two ways. First, if a file needs to be expanded, the root writes one byte of null data at the last offset. Although an exNode has a field to store the current file size, it is often recalculated based on the mappings in the exNode. Thus, it is safer to create a mapping to the last byte than to modify the size field in each exNode. Second, if a file needs to be shrunk, the mappings in the exNode are trimmed on each process. `LN_Prealloc` is implemented by having the root write 0s to the file from the next byte of the currently last offset up to the desired file size. Both functions are collective and require exNode synchronization so that the correct file size can be obtained from nonroot processes after the calls.

Atomic mode and shared file pointer have not been implemented yet. Atomic mode requires global locking either at the file system or MPI level. For MPI-IO/L, it has to be at the MPI level because IBP does not provide a locking mechanism. An implementation of byte-range lokcs using MPI-2 passive target RMA (Remote Memory Access) has been proposed [17]. However, because of the limitation in the current version of MPICH2 that requires MPI functions to be called for progress at the target, we have decided to wait until it is fixed. ROMIO currently maintains a shared file pointer by storing its value to a file. In MPI-IO/L, the shared file pointer file should be kept in a local file system, rather than a remote depot, for better performance, and thus a different handling is required. Also, the shared file pointer file should be accessed and updated atomically and requires locking.

3.3 Issues with MPI-IO Consistency Semantics

Consistency semantics define what happens when multiple processes concurrently access a common region of the file. For example, the POSIX I/O consistency semantics require strict sequential consistency, where the result of a series of concurrent I/O operations should be as if they were performed in a certain order. Moreover, it requires that each update to the file be immediately visible to other processes. On the other hand, by default, MPI-IO specifies a more relaxed set of consistency semantics more suitable for parallel I/O, and provides opportunities to optimize the performance within the MPI-IO implementation. Users can, however, select stronger consistency semantics.

The default (weak) consistency semantics in MPI-IO guarantee that when multiple processes are writing concurrently, the changes will be immediately visible to the process that wrote them, but not to the other processes right away unless explicit synchronization is performed by sync[8] or a special "atomic" mode is used. When multiple writes are performed on the same region, the only way to guarantee sequential consistency other than the atomic mode is to temporally separate them from each other by surrounding each with open, sync, or close calls. Otherwise, the result is undefined.

In MPI-IO/L each process locally updates its own exNode at writes, and `LN_Sync` synchronizes all the in-memory exNodes to make all the updates visible to every process. `LN_Sync` also combines the current content of the XML file with the in-memory exNodes to incorporate changes made by other groups of processes that concurrently opened the same file. If the other processes wrote some data to the file and called `LN_Sync`, the XML files should contain new mappings, and these should be visible. After synchronization, every process will have identical in-memory exNode, and then the root will serialize it to the XML file.

More specifically, `LN_sync` first flushes the buffer if buffered I/O is set. Then each process creates a set that contains mappings that have been modified since the last sync operation or file open if there has not been a sync yet. To identify such "dirty" mappings, we tag a newly created or modified mappings at each write. Next, we read the mappings contained in the XML file and put them into the new set. The intention is to reflect the changes made to the file by others who open the same file concurrently. If some of the mappings from the file overlap with the dirty mappings, we trim the mappings from the file so that only our changes will be visible. Next, everybody broadcasts to everybody else its dirty mappings. The mappings from the file will be further trimmed by other processes' dirty mappings, too. Then each process combines all the dirty mappings from other processes, including its own and combines them to the mappings from the file. The root will then write the combined mappings to XML files. `LN_Open` reads the mappings directly from the file, so if someone else sync'ed or closed the file before the open, it should read the changes. Close also calls sync.

MPI-IO guarantees sequential consistency within a single process. Thus, the data written by a process should be immediately visible for the following reads from the same process. With the two-phase implementation of collective I/O in ROMIO, however, the data will be shuffled among the processes to maximize the I/O performance. Thus it is likely that the data requested to be written or read by one process will actually be written or read by another process. This situation can cause problems in MPI-IO/L because of the local exNode update. Suppose that a collective write operation was issued and a portion of data that process 0 requested to write is actually written by process 1. Right after the collective write, if process 0 attempts to read the same data independently, then process 0's exNode has a mapping only to partial data and the write will not be fully visible, thereby violating MPI-IO consistency semantics. In order to prevent this problem, in MPI-IO/L, `LN_Sync` is called before each collective read operation and after each collective write operation.

[8]File open and close calls have the same effect.

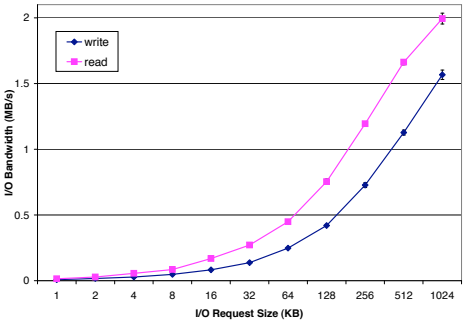


Figure 6. Small contiguous I/O.

4 Results

We conducted experiments between the Jazz Linux cluster at Argonne National Laboratory and depots at the University of Tennessee at Knoxville.[9] Jazz comprises 350 nodes, each equipped with a 2.4 GHz Pentium Xeon, either 1 or 2 GB of RAM, and 80 GB of local disk space. Jazz nodes are connected by both Myrinet 2000 and Fast Ethernet. We used Fast Ethernet to emphasize that remote I/O performance is dominated by wide-area network bandwidth rather than by the local interconnect. We expect that the performance would not vary much with Myrinet. Jazz also has 20 TB of shared disk space among all the nodes on Global File System (GFS) and Parallel Virtual File System (PVFS). The XML files were created and accessed on PVFS. All the numbers were averaged over five or more runs; the error bars show 95% confidence interval.

4.1 Synthetic Benchmark Performance

We devised a synthetic benchmark code that performs both contiguous and noncontiguous I/O operations with various parameters from a single Jazz node to a single depot.

Contiguous I/O. Contiguous direct I/O performance was measured by calling `MPI_File_write` and `MPI_File_read` for a contiguous extent of buffer. Figure 6 presents remote I/O bandwidth for small

[9]For experiments, we used pre-selected depots instead of relying on an L-Bone server.

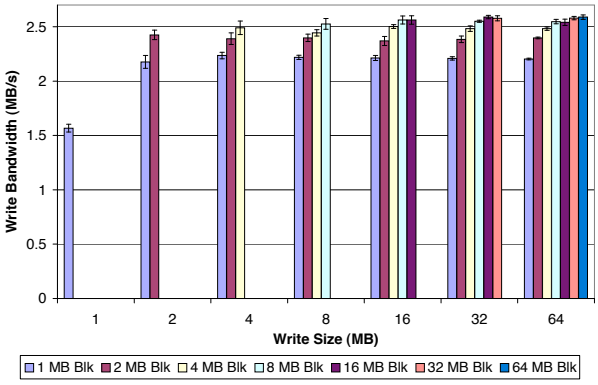


Figure 7. Large contiguous writes.

amounts of data up to 1 MB, with 1 MB block size and a single thread. The graph shows that I/O bandwidth increases as the data size increases, because the startup overhead associated with each remote I/O call dominates smaller data transfer cost. IBP maintains persistent TCP/IP connections, and for the numbers shown in this section, we excluded the first measurement that includes the connection establishment cost. The graph also shows that for small transfer, reads are faster than writes, because writes involve additional overhead for preparing a new allocation on the depot.

Figure 7 shows the remote write bandwidth for larger data, ranging from 1 MB to 64 MB, with a single thread. For each transfer, we varied the block size from 1 MB up to the transfer size, to observe the effect of block size on performance. The graph shows that the remote write bandwidth increases up to certain point (2 MB here) but remains more or less the same after that, because per-I/O request overhead becomes negligible compared to the actual transfer cost as the data size increases. Also, for each write size, larger block sizes increase write bandwidth, because with larger block size, the number of I/O calls issued to transfer the same amount of data decreases, hence the aggregate per-I/O request overhead. In the configuration used here, the performance increase becomes small after a 4 MB block size. We also ran the same benchmark with a previous version of IBP that did not maintain a pool of persistent TCP/IP connections. With this IBP, the block size effect was much bigger than what is shown in this graph, because each I/O call requires separate TCP/IP connection establishment. Another benefit of using larger blocks is that it will decrease the number of allocations created for each write and the number of mappings to them. Because of the space constraint, we show only the write performance here. The reads showed a similar trend.

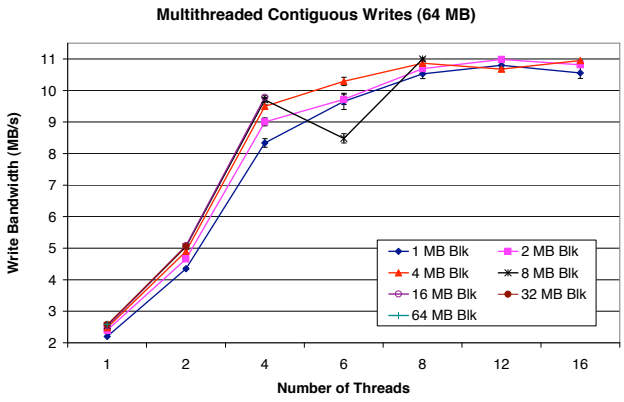
We also compared the performance of directly calling `lorsSetStore()` and `lorsSetLoad()` to the numbers shown above, to measure the overhead incurred by adding the MPI-IO and ADIO layer. The overheads observed were all less than 1% of LoRS function costs, and thus negligible.

Figure 8 shows the multithreaded I/O performance for 64 MB of data. We varied the block size from 1 to 64 MB and used up to 16 threads for concurrent transfer. As mentioned, the unit of transfer in LoRS is a block, and each thread transfers a block of data at a time. For each block size, we controlled the number of threads so that the number of threads times the block size did not exceed the amount of data to be accessed.

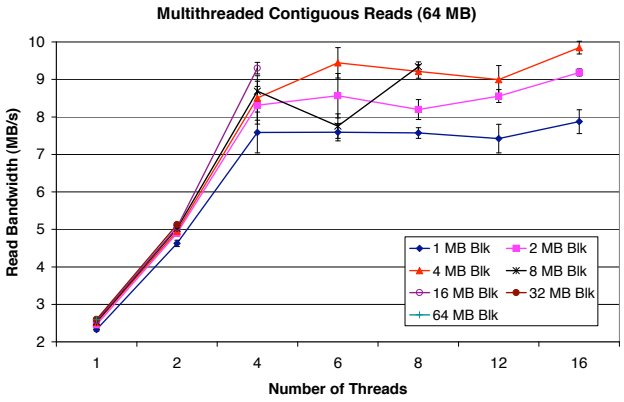
Figure 8(a) presents the threaded write performance. The graph shows that the bandwidth increases as the number of threads increases up to a certain point (about 8 threads here) and remains more or less constant after that. It also shows that with the same number of threads, larger blocks perform better. As the block size increases, however, the maximum degree of concurrency achievable decreases, and thus the overall maximum write bandwidth cannot be reached with larger blocks. In our experiments, the maximum throughput was achieved with block sizes from 2 to 8 MB. There are some exceptions where larger blocks perform worse than smaller blocks with the same number of threads, such as 8 MB block with 6 threads and 4 MB block with 12 threads. The reason is that the amount of data to be written (64 MB) cannot be a product of any of the block sizes used in the experiments and these numbers of threads. Therefore, at the last round of transfer, only a subset of threads will transfer the data, and with the larger block size the performance penalty will be higher. This situation suggests that the number of threads and the block size should be carefully chosen according to the expected amount of data access when large transfer is common.

Figure 8(b) presents the threaded read performance with the same configurations and shows similar trends to the writes. One difference is that the throughput with smaller blocks becomes much lower than with larger blocks and the same number of threads, implying that IBP depots do not handle many small concurrent reads well. On the other hand, small writes are likely to be performed efficiently with the help from the file cache, and we presume that it is true in our write tests.

Figure 9 compares the performance of buffered writes to that of direct writes. We wrote total 16 MB of data using different write sizes from 1 KB to 8 MB. Thus, if the write size is 1 KB, 16384 write operations are issued, compared to 2 writes with 8 MB size. All the writes were performed with 8 MB block size, a single thread, and a 16 MB buffer (for buffered writes). Direct write performs the same number of remote write operations as the number of write operations issued, while buffered write coalesces all the writes and performs the remote write only once. The buffered write performance shown in the graph includes the sync cost, while direct write is shown without the sync. The re-



(a)



(b)

Figure 8. Multithreaded contiguous I/O performance.

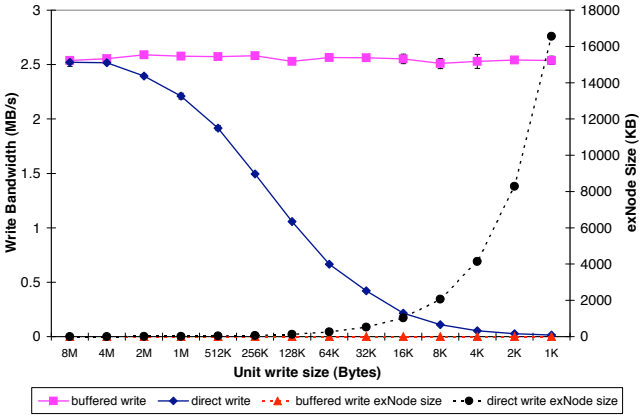


Figure 9. Buffered write vs. direct write.

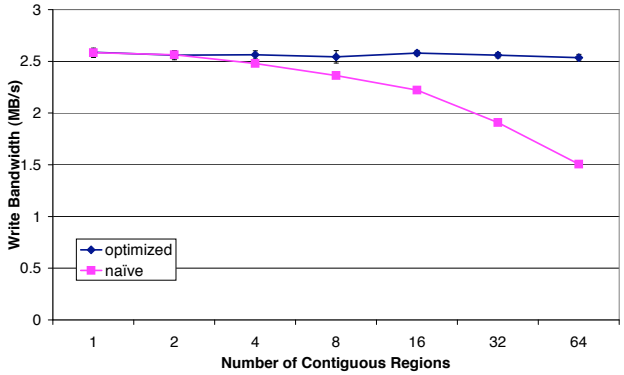


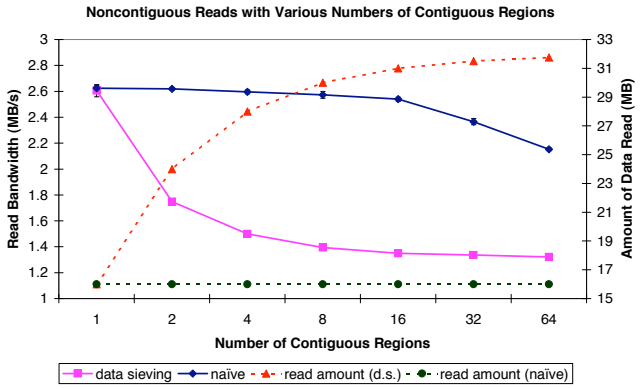
Figure 10. Performance comparison between the two noncontiguous write options.

sult shows that the direct write performance drops significantly with larger number of write operations. As mentioned, this is due to the overhead incurred by each write operations. On the other hand, the buffered write performance stays still regardless of the write size, because the actual remote write operation is issued only once in all cases. In our extreme case with 1 KB write size, buffered write performs more than 189 times better than direct write. This result suggests that buffered write should be considered when a series of sequential and contiguous writes are expected and some amount of extra memory is available.

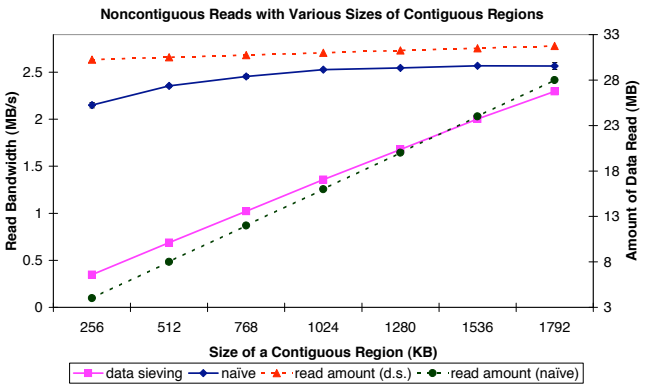
The graph also shows the size of the XML file that each configuration creates (dotted lines). Each mapping takes about 1 KB of XML code, and the size of the resulting XML file increases as the write size decreases and thus the number of remote writes and created mappings increase. With 1 KB write size, the size of the XML file exceeds 16 MB. On the other hand, buffered I/O minimizes the number of remote write performed to one, and the resulting XML file is a little bit over 1 KB. The size of XML files affects the performance of file open and sync that reads and writes the XML file. The graph does not show the effect of sync in direct I/O because the sync cost is quite small compared to the actual write cost, but the sync-only cost with 1 KB write size was more than 9 seconds, which is longer than the buffered write and sync cost combined.

Noncontiguous I/O. Figures 10 and 11 show the noncontiguous I/O performance measured with different I/O options and access patterns. In these experiments, the data is stored contiguously in memory but written noncontiguously on disk, as is common in scientific workloads. An example of this kind of access is where a 3-D array is distributed across the memory of processors in a HPF-style `(BLOCK, BLOCK, BLOCK)` fashion and a processor attempts to access its own array chunk from the file where the global array is stored in row-major order. For the experiments, the file view was first set for the strided accesses and then `MPI_File_Write` or `MPI_File_read` was called.

Figure 10 shows the performance of the two noncontiguous write approaches for 16 MB of data with a 16 MB block size and a single thread. We fixed the total amount of data to be written to 16 MB but varied the number of contiguous regions in the extent from 1 to 64. In this access pattern, the stride is twice the size of each contiguous region; that is, each contiguous region is followed by a hole of the same size. The graph shows that the optimized write performance is almost constant regardless of access patterns, because this approach issues only a small number of remote write calls (only one here) and the manipulation of mappings is done in



(a)



(b)

Figure 11. Performance comparison between the two noncontiguous read options.

memory and thus is fast. However, the naïve write performance degrades as the number of regions increases, because of the overhead incurred by many write calls. The performance gap is expected to keep increasing when finer noncontiguous access patterns are used. The size of the resulting XML file is almost the same for both approaches because the optimized approach does not reduce the number of mappings.

Figure 11(a) presents the noncontiguous read performance using the same configuration for both naïve and data sieving reads. For the reads, 32 MB of data was contiguously written to the depot prior to the access. The graph shows that both read performances decrease as the number of contiguous regions increases, but for different reasons. For the naïve approach, the performance degradation is due to the multiple I/O call overhead. On the other hand, the decreased performance for data sieving is due to the fact that the extent of the noncontiguous read (and the amount of unnecessary data read) increases with the number of contiguous regions. The graph shows the total amount of data that should be read for each approach (dotted lines).

As we keep increasing the number of regions for finer-grained read access, the data sieving performance is expected to converge to half of the 16 MB contiguous read performance (shown as the one region performance in the graph), as the total extent to be read converges to 32 MB. But, the naïve approach performance is likely to keep decreasing as we increase the number of regions, because of the excessive number of read operations issued, and it is expected to start perform worse than data sieving after some point. Thus, it is important to find the balance point and choose the faster approach according to the access patterns.

We also tested noncontiguous read performance where the data is read from a noncontiguously written file by the noncontiguous write test above. In this case, however, data sieving loses its benefit for finer accesses because now a separate mapping exists for each contiguous region and the data sieving will have to issue multiple read requests, one for each mapping, even though it wants to read a contiguous extent from the logical file. This is still true when noncontiguous data was written to only a small number of allocations using the optimized method. Although several contiguous regions were packed and written to a single allocation, multiple read requests have to be issued to access them, because the current exNode structure does not contain the information on how the data was written.

Figure 11(b) shows the performance from another noncontiguous read test, where we fixed the number of contiguous regions to 16 and the stride to 2 MB and varied the size of each region from 256 KB to 1.75 MB. Unlike the previous tests where a constant amount of data was read, here the amount of requested data increases as the size of each region increases. The graph shows both the apparent read bandwidth (solid lines) and the amount of data that must be read to perform the operation (dotted lines). For the naïve read, the amount of requested data and the amount of read data are the same, and its performance slightly increases as the region size increases, because of the diminished effect of the I/O call overhead for larger transfers. For data sieving, regardless of the region size, the amount of actual read is almost constant, close to 32 MB, and thus the apparent read bandwidth is low with smaller regions and keeps increasing for larger regions. But, it did not catch up with the naïve read performance in the region sizes used in our experiments. This result again confirms that the deterministic factor between data sieving read and naïve read is the number of I/O calls to be issued and the amount of unnecessary data read.

4.2 Tiled I/O Performance

To better evaluate the MPI-IO/L performance for real scientific applications, we used the tiled I/O benchmark, which implements representative I/O access patterns common in many parallel scientific applications—parallel I/O for distributed multidimen-

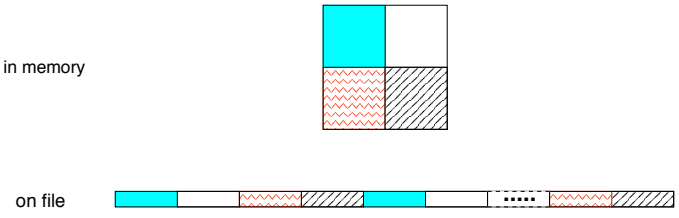


Figure 12. A 2×2 tile mesh and its file representation.

sional arrays. Its name comes from the fact that the benchmark simulates the I/O operations needed to visualize a single image file with an array of displays (or *tiles*) for higher-resolution playback on a larger screen, but it can also be used for writes. In the benchmark, tiles are created by dividing a 2-D data set along each dimension, and each tile belongs to a compute process. The tiles are written to and read from a global row-major order file.

In our experiments, 4096×4096 arrays with 4-byte elements were distributed across a 4×4 tile mesh. Each tile is identical in shape (1024×1024) and size (4 MB each), totaling 64 MB of data per array. The original tiled I/O benchmark lets users choose between collective and noncollective I/O for the array file access. We modified the code so that different noncontiguous I/O options can be chosen, too.

The first set of experiments measures the tiled I/O performance between 16 Jazz nodes and the same IBP depot used for previous experiments. Each process accesses files by direct I/O with a single thread and 4 MB block size, the same as the tile size.

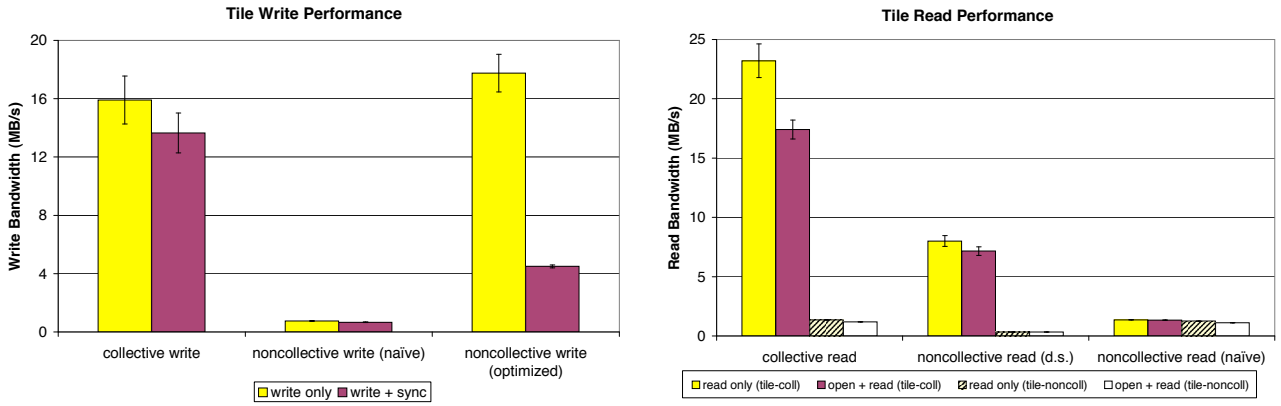
For the writes, we tested both collective and noncollective approaches. As mentioned, collective I/O optimizes parallel I/O performance by issuing fewer, larger contiguous I/O requests using the knowledge of the global data distribution. Data reordering among processes is typically required for this. The tiled I/O benchmark accesses the entire array distributed across multiple processes and thus is a good fit for collective I/O. Since the entire extent of the array data does not contain any holes, collective writes can be carried out by using contiguous writes only. Without collective I/O, each process should write its own tile data (a subarray) to the common file, and this procedure requires noncontiguous writes. Figure 12 shows how tiles in a 2×2 mesh are stored in a row-major-order file. Since rows in each tile are distributed across the whole extent of the file, accessing a tile will be noncontiguous. But, unlike collective I/O, communication for data reorganization among processes will not be required. We used two noncontiguous write options available in MPI-IO/L: naïve and optimized.

Figure 13(a) presents the tile write performance with different I/O approaches. The graph shows two bars for each write option. The first shows the bandwidth of the write operation only, while the second one shows the bandwidth with the sync cost included. This is reasonable because in many write-oriented simulations different data sets are stored in different files and sync cost is visible when closing each file. With collective writes, the tile data is first shuffled among 16 processes to optimize write access patterns, and then each process concurrently issues a 4 MB write request, resulting in 16 mappings and allocations. Even though the local communication cost is included, the collective write performance is better than that of the single process write with 16 threads shown earlier, because the data transfer is performed via multiple network interface cards and also it does not incur the context switching costs. With the sync cost included, the collective write bandwidth drops about 14%. Since the sync cost depends on the number of mappings in the exNode, if we collectively write larger data with the same number of mappings by increasing the block size, the performance drop caused by the sync is expected to decrease more.

With the noncollective write with the naïve approach, performance drops significantly. Each process issues 1024 remote write requests (1024 rows in the subarray), each of which writes $1024 \times 4 = 4096$ bytes of data, and thus the overhead associated with multiple write requests adds up and hurts the performance. This will create 16384 (1024×16) mappings in the exNode, and the resulting XML file will be over 16 MB, 1024 times larger than the XML generated for collective write. This large exNode significantly increases the sync cost. Indeed, in our experiments, the sync for noncollective writes took longer than twice of the collective write and sync cost for the same data set.

With the optimized approach, the write-only bandwidth becomes about 12% higher than that of the collective write because it also performs only 16 remote writes (one per each process) like the collective write but does not involve local communication cost. It does include the cost to create 1024 mappings on each process, but that seems negligible. With the sync cost, however, the write bandwidth becomes almost one-fourth of the write-only bandwidth, because optimized writes create the same number of mappings as the naïve approach does and thus sync becomes expensive for the exNode with 16384 mappings.

Therefore, even with slightly worse write-only performance than optimized noncontiguous writes, collective write seems to be a clear winner among the three approaches for the following reasons. First, collective write reduces the overhead incurred by each write call invocation by issuing fewer larger write operations. Second, since it only issues fewer write requests, fewer mappings are created, and thus it decreases the cost associated with sync. More benefits that collective write can bring are described below.



(a) (b)

Figure 13. Tile I/O performance.

The tile read performance was also measured for both collective and noncollective read, and both data sieving and naïve noncontiguous read were used for noncollective reads. We also evaluated the read performance from two network files, one written collectively (`tile-coll`) and the other written noncollectively with optimized noncontiguous writes (`tile-noncoll`). The former file contains 16 mappings to 16 different allocations, while the latter contains 16384 mappings to the same number of allocations (1024 mappings to each allocation).

Figure 13(b) shows the tile read performance. In this graph, we show the read bandwidth with and without file open cost, to see the effect of opening a file with a large number of mappings. We did not include the sync cost here because it is negligible when the file is open for read only. The graph shows that the collective read performance with `tile-coll` yielded the best performance among the configurations used. This is because collective read issues fewer, larger remote read requests and thus reduces the effect of I/O call overhead. When the open cost is included, the bandwidth drops about 25%. As mentioned above, since the open cost is proportional to the number of mappings that the exNode contains, the gap between the two bars will decrease if we read larger data with the same number of mappings.

When the same collective read is performed on `tile-noncoll`, however, the read bandwidth significantly decreases. Compared to the collective read with `tile-coll`, a factor of 17 performance decrease was observed. This is because even though collective read issues larger read requests, each large read request is translated into many more small remote reads because of the way `tile-noncoll` was written. Also, the cost of opening `tile-noncoll` is more than twice the collective read cost of `tile-coll`. When this open cost is included, the performance drops further, although not obvious in the graph. This result again confirms why collective writes can improve I/O performance in MPI-IO/L, in this case, by reducing the number of mappings in the exNode. The way how a file was written affects not only the sync performance but also future open and read performance.

When `tile-coll` was read noncollectively with the data sieving, about one-third of the collective read bandwidth was achieved, because the data sieving reads more data than what is requested. In our configuration, the size of each tile is 4 MB, but the data sieving read reads almost 16 MB of data on each process and then picks out the requested tile data out of the data read. Moreover, when the whole array is read with the data sieving, each tile is read almost four times, while the collective read can effectively avoid this redundancy. Thus, although the data sieving issues larger read requests (like the collective read), it is not a desirable option for distributed global array reads, especially with slow remote I/O, and hence its use should be restricted for true noncontiguous I/O patterns. Reading `tile-coll` noncollectively with the naïve approach performs even worse than the data sieving. The naïve approach issues 1024 remote read requests on each process (16384 in total), and the per-I/O call overhead for small reads hurts the performance too much. This is an example where the data sieving works better than the naïve approach, but both approaches perform far worse than collective reads. When the open cost is included, the performance drops about 10%.

When `tile-noncoll` is read noncollectively, however, the data sieving performs much worse than the naïve approach. The naïve approach issues the same number of remote read requests that it issues for reading `tile-coll`, resulting in similar performance. However, since the data sieving reads extra data and these extra read requests are translated into many small remote reads because of the way the file was written, more overhead is incurred and eventually the performance decreases further. In our configuration, almost four times more read requests that read 4096 bytes remotely are issued for the data sieving than for the naïve approach, which issues 1024 read requests

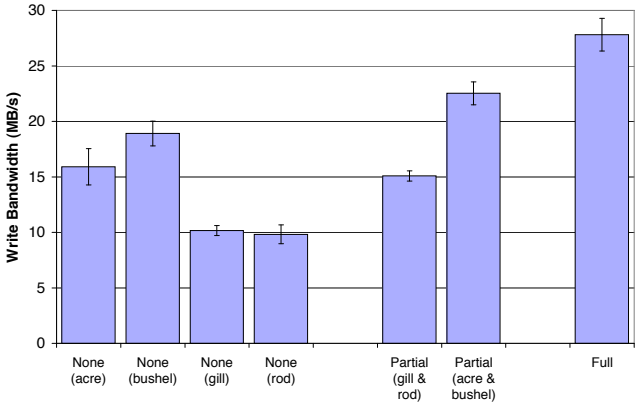


Figure 14. Effect of reordering depots in the depotpool on collective write performance.

on each process. For systems like IBP depots where the overhead for small I/O request is higher, it is critical for performance to reduce the number of remote I/O requests. We note that even though larger read requests are made, they could issue many small read requests according to how the file was originally written.

For data sets with higher dimensions, this performance discrepancy between reading collectively and noncollectively written files is likely to increase because, with higher dimensionality, more complex noncontiguous access patterns that contain many more contiguous regions are likely to be expected. Thus, the benefit of reducing the number of mappings created becomes even more important.

When the tile data is accessed from multiple depots, better I/O performance is expected than with a single depot, because multiple machines and disks are handling concurrent data streams and performing disk I/O. As discussed, however, even with multiple depots, if we do not carefully reorder depots in the depotpool, multiple write requests can be concentrated to a single or a small number of depots and achieve worse performance than what is potentially possible.

Figure 14 shows how the depot reordering can affect collective write performance. We used four IBP depots, named acre, bushel, gill, and rod, to collectively write 64 MB of data from 16 processors on Jazz. The graph shows three groups of bars. The first group shows the write performance where no depot reordering was performed and thus each processor on Jazz was given the same list of depots. In this case, all the processors start with the first depot in the list and perform the write on the same depot even though multiple depots are available in the list. Each bar in the first group shows the write-only bandwidth when each of the four depots is placed at the beginning of the list. The graph shows two relatively faster depots (acre and bushel) and two slower depots (gill and rod).

The second group shows the performance where the depot list is partially reordered. In this configuration, half of the processors are given a depot list and the other half another list where depots are ordered differently from the first list. The graph shows the performance where two slow and two fast depots appear first in the lists. In both cases, the performance increases about 20–50% compared to that of the faster depot between the two. The third group shows the result with the full reordering, where all the four depots appear first in the lists and a group of four processors is given each list. With this full ordering, the performance increases about 47% of the performance with bushel only, the fastest one. This performance was achieved even though slower depots are used for the writes. When very slow depots are used, however, the performance may drop with the full ordering. For further optimization, therefore, reordering should consider the performance of each depot. For remote reads from multiple replicas, LoRS implements dynamic load balancing among the participating depots [13].

5 Discussion

Logistical Networking was not originally designed for parallel I/O. Typical users run command-line tools to access remote files, and the original design works well for such uses. For parallel I/O, however, its current design imposes limitations that might seriously affect the performance. This section discusses such issues and suggests how we can improve Logistical Networking to make it work better with parallel I/O systems.

Downsizing exNodes. LoRS currently creates a new allocation and a mapping to it for each write. Thus, fine-grained writes or frequent updates on the file could result in very large exNodes. With larger exNodes, open, sync, and close, which access the XML representation of exNodes, become more time-consuming. Large exNodes also take more memory on each process when stored in in-memory structures and thus reduce the amount of memory that can be used for storage, buffering, and caching of data. Moreover, it is not convenient to share such bulky exNodes with other people.

One can use different ways to downsize the exNodes. First, the number of mappings generated by applications can be reduced by observing the access patterns and eagerly using the available optimizations. For example, buffered I/O and collective I/O are useful to coalesce small I/O requests and thus reduce the number of actual writes. Also, using a larger block size can reduce the number of allocations and mappings generated for each write. These approaches will also bring better performance by reducing the overhead associated with each I/O call.

One can also change the in-file exNode representation. XML is easy to understand and flexible but takes much space because it is text-based. ExNodes can be described more compactly if represented in binary for-

mat. For example, exNodes can be serialized to a file as they are stored in memory. For this, a certain order to store each field and special tags to distinguish one field from another might be required. A tool that parses the binary representation and converts it into XML or other user-friendly format could be provided, too.

Another way to downsize the exNodes is to keep the XML format but rearrange the order of fields appearing in the file. Currently, each mapping is listed separately, regardless of the allocation it points to. With our optimized noncontiguous writes, however, multiple small writes can be packed into few bigger ones, and thus multiple mappings can now point to the same allocation. Mappings to the same allocation have identical values for certain fields, such as the three long capability key strings, and with the current representation, the same values are repeated redundantly. A better way to avoid such redundancy is to introduce a hierarchy between allocations and mappings so that for each allocation, allocation-related information such as capability keys appears first and then a list of mappings associated with the allocation. In addition to reducing the exNode size, this approach provides an opportunity for noncontiguous read optimization, as described below.

The implementation of the write-at-offset IBP can also reduce the size of the exNode. Currently, for an overwrite to an existing allocation, the overwritten portion is written to a new allocation, resulting in additional mappings. The write-at-offset IBP will allow the overwrites to be performed on the existing allocation itself and thus will not increase the number of mappings.

Finally, off-line data rearrangement can reduce the number of mappings by contiguously rewriting noncontiguously stored data to new allocations. This approach resembles the disk compression utility and requires extra network bandwidth and storage space.

Efficient Noncontiguous I/O Support. MPI-IO/L's optimized noncontiguous writes pack and write contiguous regions in a noncontiguous extent into a single allocation. But, because of the limitation of the current exNode representation, the future reads whose extents overlap with such noncontiguously written extent cannot detect this fact and are not optimized properly. Instead, they are performed by issuing separate small read requests to the same allocation. Another example where multiple mappings point to the same allocation is the overwrite to an existing allocation. If the middle portion of the allocation happens to be overwritten, it will result in three mappings, one that points to the new allocation for the overwritten portion and two that point to the head and tail portions of the original allocation.

If the exNodes are organized as proposed above where a set of mappings that point to the same allocation is grouped under the allocation information, read could be further optimized as follows. First, we identify the allocations that overlap with the requested read extent. Next, for each such allocation, we determine whether contiguous or noncontiguous read needs to be performed by examining the mappings that point to the allocation. If noncontiguous read is detected, then data sieving could be performed within that allocation for better read performance. Since this approach performs data sieving on each allocation, instead of for the whole noncontiguous read extent, it could reduce the amount of extra read significantly. Especially if a noncontiguously written file will be read again noncontiguously with the same or similar access patterns (e.g., noncollectively reading `tile-noncoll` in the previous section), such noncontiguous read operations can be carried out with contiguous reads.

If noncontiguous I/O is supported at the IBP level, further optimization is possible. For example, list I/O [16] allows the description of a noncontiguous I/O access using a single I/O interface, reducing the per I/O-call overhead. Similarly, datatype I/O [7, 10] provides an interface where a noncontiguous I/O pattern is conveniently described using a MPI-derived datatype. Once the underlying file system understands such advanced interfaces, it can better optimize the noncontiguous I/O performance.

Asynchronous I/O. Asynchronous remote I/O is another desired feature in Logistical Networking. Considering that the available network bandwidth to many users is still often low, asynchrony in remote I/O could provide a powerful optimization for many applications by hiding remote I/O latency through write behind or prefetching [10]. Particularly, asynchronous I/O could be useful when an application needs to replicate its output data. In such a case, the primary copy can be written synchronously, but the other copies can be created asynchronously, not stalling the application further.

6 Related Work

A few efforts have provided remote I/O through MPI-IO, all using ROMIO as the testbed. RIO [9] provided a preliminary design and proof-of-concept implementation of remote file access in ROMIO. RIO ensured portability by using the ADIO layer and later work, including MPI-IO/L, followed the same approach. However, RIO required a certain processor configuration that could cause inefficiency and relied on a legacy communication protocol. RFS [10] is a recent work that removed RIO's shortcomings. RFS seeks to reduce the apparent remote write cost by overlapping writes with subsequent computation phases through aggressive memory buffering. Both RIO and RFS adopt a client-server architecture, where a remote I/O request from the client is shipped to the server and executed there. On the other hand, MPI-IO/L translates

each remote I/O request into LoRS calls and relies on Logistical Networking to take care of data transfer and storage. Remote I/O using GridFTP for ROMIO [3] uses a similar approach.

Like Logistical Networking, specialized data transfer and storage services such as GridFTP [2], Kangaroo [14], and GASS [6] can be used as a means of wide-area data transfer for I/O libraries. These mechanisms provide useful remote I/O features such as secure communication, a chainable server architecture, and workload-specific I/O optimizations. A metadata catalog system such as the MCAT in the Storage Resource Broker (SRB) [4], which is used to identify and discover resources and data sets of interest using their attributes instead of physical file names, is another useful feature to have in an I/O system. For Logistical Networking, a simple, Web-based catalog system called Logistical Distribution Network (LoDN) was built recently. The use of LoDN together with MPI-IO/L will further improve the usability of the system.

7 Conclusions and Future Work

We have presented the design and implementation of MPI-IO/L, an efficient remote I/O support for MPI-IO using Logistical Networking. Leveraging Logistical Networks, MPI-IO/L enables high-performance remote I/O and provides a flexible way to describe network files and to share them with other people. Our implementation with ROMIO provides various options for basic I/O operations and the performance study shows that each approach has its own advantages and disadvantages. We currently let the user choose the I/O methods that work the best with her own application's I/O needs. We have also identified a number of areas in which Logistical Networking could be improved to better suit the needs of parallel I/O.

Future work includes the implementation of atomic mode and shared file pointers, improved buffered I/O that buffers multiple extents, noncontiguous I/O performance analysis via performance models, and evaluation with real applications on production scales.

8. Acknowledgment

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

[1] NCSA HDF home page. http://hdf.ncsa.uiuc.edu.

[2] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, 2002.

[3] T. Baer and P. Wyckoff. A parallel I/O mechanism for distributed systems. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[4] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of the Annual IBM Centers for Advanced Studies Conference (CASCON)*, 1998.

[5] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *Proceedings of SIGCOMM*, 2002.

[6] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[7] A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp. Efficient structured access in parallel file systems. In *Proceedings of the International Conference on Cluster Computing*, 2003.

[8] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[9] I. Foster, D. Kohr, Jr., R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1997.

[10] J. Lee, X. Ma, R. Ross, R. Thakur, and M. Winslett. RFS: Efficient and flexible remote file access for MPI-IO. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[11] J. Li, W.-K. Liao, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC03*, 2003.

[12] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Standard*. 1997.

[13] J. S. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207–224, 2003.

[14] D. Thain, J. Basney, S.-C. Son, and M. Livny. The Kangaroo approach to data movement on the Grid. In *Proceedings of the Symposium on High Performance Distributed Computing*, 2001.

[15] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[16] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[17] R. Thakur, R. Ross, and R. Latham. Implementing byte-range locks using MPI one-sided communication. In *Proceedings of the European PVM/MPI Users' Group Meeting*, 2005.

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.